JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. 2000; **12**:25–45

Research

A maintenance-oriented approach to software construction

Stephen R. Schach[1,*,†] and Amir Tomer[2]

[1]*Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville TN 37235, U.S.A.*
[2]*Computer Science Department, The Technion – Israel Institute of Technology, Haifa 32000, Israel*

SUMMARY

Maintenance is performed not only after a software product has been delivered to the client. On the contrary, the requirements frequently change during development, thereby necessitating reconstruction of the artifacts that have been developed to date. In this paper we present a process for software construction that recognizes maintenance as an essential aspect of the entire life cycle of the software product, considered from the very first steps of the initial development. The process may be used in conjunction with any software development or maintenance methodology. Our process consists of two components: a procedure that is uniformly applied at every step of the chosen methodology, whether development or maintenance; and a data structure, the propagation graph, which is updated at every step. When requirements change, the propagation graph is used to determine which artifacts of the software product are impacted by the change in requirements. From the viewpoint of changes in requirements, the process treats development as a special case of software maintenance. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: software artifacts; requirements allocation; change propagation; software development; software requirements; requirements traceability

1. INTRODUCTION

A wide variety of software development methodologies are available for constructing a software product. However, almost all such methodologies virtually ignore maintenance. This is a serious problem because, on average, about 67% of software budgets are spent on maintenance [1]. Furthermore, there are many organizations that devote as much as 80% of their time and effort to

*Correspondence to: Dr. Stephen R. Schach, Department of Electrical Engineering and Computer Science, Vanderbilt University, Box 1679 Station B, Nashville TN 37235, U.S.A.
†Email: srs@vuse.vanderbilt.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: CCR-9900662

Received 9 August 1999
Revised 19 November 1999

maintenance [2]. Without the assistance of the methodology that was utilized to develop the product, maintainers frequently fall back on the intuitive methods that had to be used before methodologies became available, often with most unsatisfactory results.

In fact, maintenance is frequently required long before the product is delivered to the client. Software developers usually have to cope with continually changing requirements and evolving technologies, the so-called 'moving target problem' [1]. That is, developers may have to perform software maintenance, sometimes as early in the life cycle as shortly after the requirements have been approved by the clients. Furthermore, the need for maintenance often recurs throughout the development process, notwithstanding the fact that, as previously stated, few development methodologies can handle such maintenance tasks well.

In an earlier paper, Tomer and Schach [3] presented the evolution tree life-cycle model. This model explicitly describes the way a software product evolves as the requirements change.

In this paper we present a maintenance-oriented approach to software maintenance and construction. We put forward a process that incorporates maintenance into any stepwise development methodology. Unlike other work in the field such as that by Li and Offutt [4], our process focuses on analysis and design. Our process is based on a procedure that is applied to every step of a stepwise methodology, irrespective of the size of the software product. For example, the procedure can be applied to the complete design phase as well as to the detailed design of just one module or class. The key to our process is a graph that displays dependencies between requirements and the artifacts of the methodology. Then, when a requirement changes, the graph is used to propagate the change throughout the product. In this way, the developer or maintainer is alerted as to which portions of the product need to be checked and, if necessary, changed to ensure that the product correctly implements the current requirements.

In accordance with IEEE Standard 610.12, maintenance is frequently viewed as an activity that is performed only after the software product has been delivered to the client [5]. On the other hand, the fact that maintenance may be required throughout the development process is recognized in ISO/IEC Standard 12207, which states that the maintenance process is activated when 'software undergoes modifications to code and associated documentation due to a problem or the need for improvement or adaptation' [6]. In this paper we use the term 'maintenance' as defined in ISO/IEC Standard 12207.

In our figures we use UML as the notation for object-oriented analysis and object-oriented design [7]. We also use UML as a notation to clarify the definitions we introduce later in this paper.

In Section 2 we introduce a software construction example, and in Section 3 we show what additional information needs to be recorded while carrying out each step of the methodology chosen by the developer. Then, in Section 4 we revisit and enhance the example and show how the graph created during initial construction is used to determine which parts of the example need to be changed. Finally, in Section 5 we present our results and conclusions.

2. IMPLEMENTING REQUIREMENTS

2.1. Requirements propagation

In this section we describe how the client's requirements are implemented in terms of software artifacts. In particular, we show how the requirements are propagated from phase to phase.

A software product is engineered to satisfy the client's requirements. This is a progressive process, typically comprised of the following phases:

- requirements, where the client's requirements are elicited;
- analysis, where requirements are analysed, resulting in a specification document;
- design, where a solution is designed, resulting in a design document; and
- implementation, where software modules are programmed, resulting in the code.

The final product, in the form of an executable program, is then validated to ensure that it meets the client's original requirements. Throughout its entire life cycle a software product is likely to go through changes, driven by the need for correction, perfection or adaptation. These maintenance activities are performed using the same phased process described above, except that the intermediate products are not generated from scratch, but instead are modifications of existing versions.

During this process the original requirements do not merely propagate through the various phases. Rather they are elaborated in each phase in order to enable the next phase to be accomplished. This requires using relevant methods and languages; for example, data flow diagrams (DFD) for the analysis, pseudocode for the design and a programming language for the implementation.

During the analysis phase the software architecture is constructed, based on the *client's requirements*. However, choosing a specific architecture imposes on the design a set of additional requirements—the *specification requirements*. Next, the design phase is expected to implement both the previous sets of requirements (namely, the client's requirements and the specification requirements) in the form of a design document, resulting in a further additional set of requirements, the *design requirements*. A typical example of a design requirement is a specific data structure that must be used. Finally, all the above requirements (namely, the client's requirements, the specification requirements and the design requirements) must be implemented in the final product in the form of source code. Requirements, therefore, are built up in layers, as depicted in Figure 1. This layered arrangement of requirements is supported also by requirements CASE tools, such as Rational RequisitePro® [8].

The description above views the course of software construction as a sequence of phases in which the products of each phase (a) implement the requirements of the previous phase, and (b) specify the requirements for the next phase. This specification-implementation chain is discussed in detail in Tomer [9] and Tomer and Hogger [10].

The various products of each phase are here referred to as *software artifacts*. The artifacts are viewed through *software documents*, as will now be described.

2.2. Software artifacts

A *software artifact* is a tangible piece of information obtained through the software development process. The concept of an artifact is included as part of the Unified Software Development Process [11], where a formal definition of the term may be found. Although originating in object-oriented analysis and design, the concept may be applied to other software engineering development paradigms.

In practice, a software artifact is generated for the purpose of implementing a subset of the requirements. Thus, each artifact is associated with a set of requirements that have been allocated to it. The requirements allocation is not necessarily one-to-one—that is, a single requirement may be allocated to more than one artifact. Although the Unified Process does not directly assign each artifact to a specific development phase, the common practice of software engineering tends to associate certain

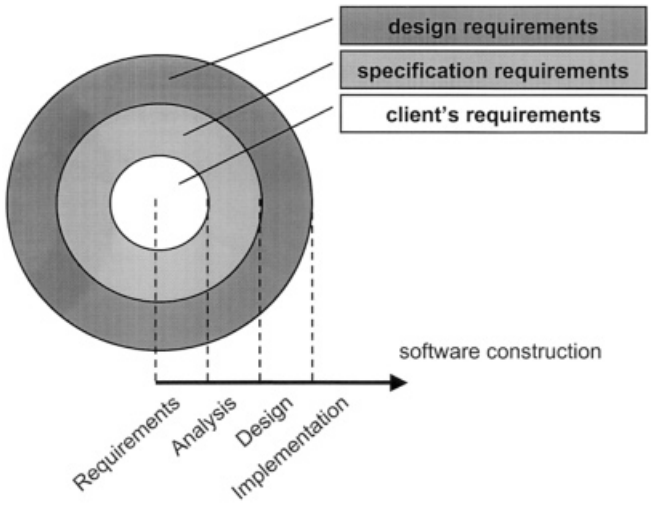


Figure 1. Requirements build-up during software construction.

artifacts with certain phases. For example, DFDs are associated with analysis, physical component diagrams with design, and source-code modules with implementation. However, certain artifacts may be utilized in several consecutive phases. For example, a class is usually constructed during analysis, where its data properties and relations are specified. Later, during design, the same class is revisited, and methods are defined so that it can carry out its required activities. Finally, this same class is used for automatic code generation during implementation.

The association of a set of requirements with an artifact, for the purpose of requirements traceability [12–14], does not by itself reveal the way these requirements are implemented by that artifact. Moreover, such an association does not prove that the artifact indeed satisfies these requirements. Such approval is usually achieved through a verification review, whose role is to check that all the allocated requirements are satisfied. The common practical way of verification (for non-executable artifacts) is through some kind of review, in which qualified personnel obtain an understanding of the artifact and the way it implements the requirements. For example, in order to check that the requirement 'messages are passed from senders to receivers' is met by an artifact, it may be necessary to show that a 'passing' function exists, and that it is associated with two entities—'sender' and 'receiver.' The nature of this check depends on the specific type of the artifact and the language in which it is expressed, as for example, a DFD process, a UML class specification or a Java source module.

Thus, the relationship between artifacts and requirements is twofold:

- An artifact implements one or more requirements (that originated in artifacts of previous phases).
- An artifact generates zero or more new requirements (to be implemented by artifacts during successive phases).

Using UML, these two relations are illustrated on the left side of Figure 2.

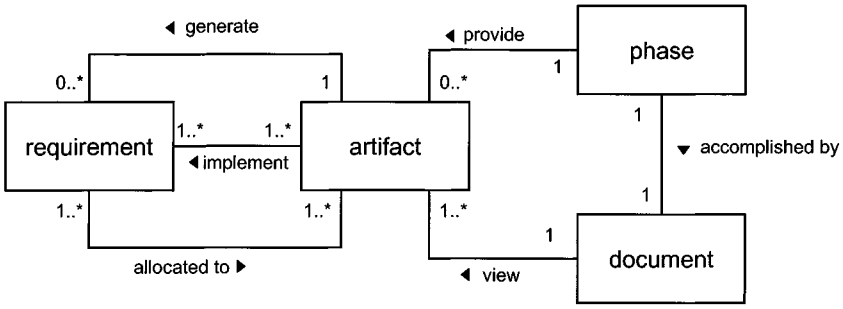


Figure 2. Requirements, artifacts, phases and documents and their relationships.

2.3. Software documents

The formal deliverables of a software product (other than the executable software itself) are *software documents*. Because artifacts may be generated and maintained in computerized forms by using CASE tools, a document usually expresses the contents of artifacts in a written language (formal and/or informal) and/or in diagrams. Each construction phase is usually terminated by submission of a document, namely, a specification document, a design document, or documented source code. Each document, therefore, is associated with a set of artifacts whose contents are encapsulated within the document. However, as stated above, certain artifacts may be associated with more than one phase. The relationship between artifacts, documents and development construction phases (illustrated using UML on the right side of Figure 2) is as follows:

- each phase provides (i.e., generates or modifies) zero or more artifacts;
- each phase is accomplished by a document; and
- each document is a view of one or more artifacts.

Although the documents are not directly associated with the requirements, but rather with the artifacts, they are the practical source through which requirements are tracked during the software life cycle.

2.4. Requirements management and configuration management

Throughout its life cycle, a software product is controlled and managed to ensure that its requirements are consistently satisfied. Moreover, each version of the product—that is, each specific configuration of its artifacts—is controlled and managed to ensure that it satisfies a given specific set of the client's requirements. These two management processes are addressed as level 2 key process areas (KPAs) in the SEI capability maturity model (CMM) [15] and are summarized below.

The requirements management KPA defines the means for stipulating the software baseline as a set of 'system requirements allocated to the software', and for assuring that this baseline is satisfied during the product's development and maintenance. The software configuration management KPA requires

that changes be incorporated into software products under a controlled process, in accordance with the baseline requirements, without specifying how the baseline requirements propagate into the specific software products (artifacts).

Our requirements propagation approach, which was introduced above and is elaborated below, bridges this gap. Our approach provides a practical method for incorporating the client's requirements and their derivatives into the software artifacts, with full control over the allocation of specific requirements to specific artifacts. In more detail, we follow the concept of 'requirements allocation' of the requirements management KPA and start with the system (or client's) requirements allocated to the software [product]. Then we take this approach further, applying it to the software requirements allocated to the specification [artifacts], the specification requirements allocated to the design [artifacts] and the design requirements allocated to the implementation [artifacts]. This also emphasizes, as will be elaborated later, that changes in the configuration are caused not only by changes in the requirements, but also by changes in requirements *allocation*.

Existing configuration management tools [14] usually do not adequately support all aspects of requirements propagation, as described above. More specifically, configuration management tools handle dependencies between modules, i.e., source code dependencies. Thus, configuration management tools support propagation of changes within source code. In general, however, they do not support propagation of changes caused by changes in the requirements, let alone in requirements allocation. That is, configuration management tools rarely support propagation of changes in artifacts other than source code.

2.5. Example 1

In this section we present a detailed example of software construction that highlights how the client's requirements are implemented via the various artifacts, from specification to design and then to implementation. The emphasis is on requirements propagation and the association of requirements with artifacts.

2.5.1. Requirements

A software system needs to process messages arriving from some external source. The client's requirements (denoted by ClientReq_i) are as follows:

- ClientReq_1: messages arrive randomly;
- ClientReq_2: messages shall be processed in order of arrival;
- ClientReq_3: the peak rate is 300 messages per second.

In the following we show how these requirements are implemented during each phase. Here we use objected-oriented analysis and design with the UML formalism.

2.5.2. Analysis

Object-oriented analysis of ClientReq_1 and ClientReq_2 shows the need for three classes:

- a *message* class, to provide for the contents of messages;

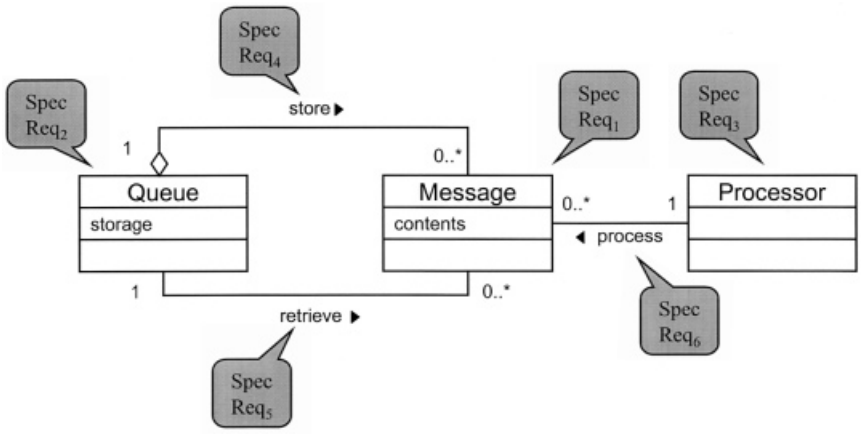


Figure 3. The specification artifacts of Example 1 in UML format.

- a *queue* class, to administer message storage and retrieval in the desired order;
- a *processor* class, to handle message processing.

In addition, the relations between these classes need to be specified by means of a class diagram. Therefore, we need four new specification artifacts as follows:

- $SpecArt_1$: a Message class specification (from $ClientReq_1$ and $ClientReq_2$);
- $SpecArt_2$: a Queue class specification (from $ClientReq_1$ and $ClientReq_2$);
- $SpecArt_3$: a Processor class specification (from $ClientReq_2$);
- $SpecArt_4$: a class diagram, specifying the relations between these classes (from $ClientReq_1$ and $ClientReq_2$).

The class diagram of Figure 3 demonstrates these artifacts in UML notation. The specification artifacts generate an additional set of requirements, the specification requirements, denoted by $SpecReq_1, \ldots, SpecReq_6$, as follows:

- $SpecReq_1$: a Message class shall exist, with contents attribute (from $SpecArt_1$);
- $SpecReq_2$: a Queue class shall exist, with storage attribute (from $SpecArt_2$);
- $SpecReq_3$: a Processor class shall exist (from $SpecArt_3$);
- $SpecReq_4$: a queue shall store zero or more messages (from $SpecArt_4$);
- $SpecReq_5$: a queue shall retrieve zero or more messages (from $SpecArt_4$);
- $SpecReq_6$: a processor shall process zero or more messages (from $SpecArt_4$).

There are two reasons why we have labelled these classes as specification requirements ($SpecReq_i$). First, the need for these requirements initially became apparent during the analysis (specification) phase. Second, these requirements must be satisfied by the design and implementation phases.

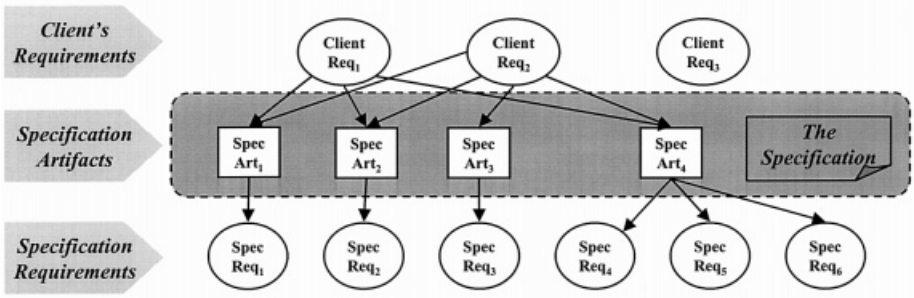


Figure 4. The specification artifacts (rectangles) and their associated requirements (ellipses).

In addition, the relations between these classes (as specified by the labelled links between the classes in Figure 3) also constrain the design regarding the interfaces between the classes. Therefore, they should also be recorded as specification requirements.

The relations between the artifacts and their associated requirements may be presented graphically, as shown in Figure 4. Ellipses represent requirements, whereas rectangles represent artifacts. An arrow from a requirement to an artifact denotes that this requirement was implemented by that artifact, whereas an arrow from an artifact to a requirement denotes that the requirement was generated by the artifact.

It should be noted that ClientReq_3 has not been implemented in the specification. ClientReq_3 is a typical performance requirement that is likely to be provided for during the design phase. Another observation is that requirements (ClientReq_1, for one) may be implemented by more than one artifact.

Finally, we expect a specification document to be issued at the end of the analysis phase, displaying the information stored in all the specification artifacts together with any necessary additional documentation. A traceability matrix, documenting the relations presented in Figure 4, is a necessary component of any such specification document.

2.5.3. Design

During design, the mutual behavior of the objects is investigated, yielding a sequence diagram (see Figure 5). As a consequence, existing classes are assigned methods according to the services they need to provide to other objects during execution.

The design artifacts, which are assigned to implement the specification requirements, are based on the classes introduced in the specification, enhanced with the definition of methods and data structures, as become apparent during design. In practice, one could use the analysis class models for this enhancement. However, for the clarity of our example we will treat the design classes as new artifacts, denoted as design artifacts (DesignArt_i), as follows:

- DesignArt_1: the message class design (from SpecReq_1);
- DesignArt_2: the queue class design (fromSpecReq_2, SpecReq_4, SpecReq_5 and ClientReq_3);

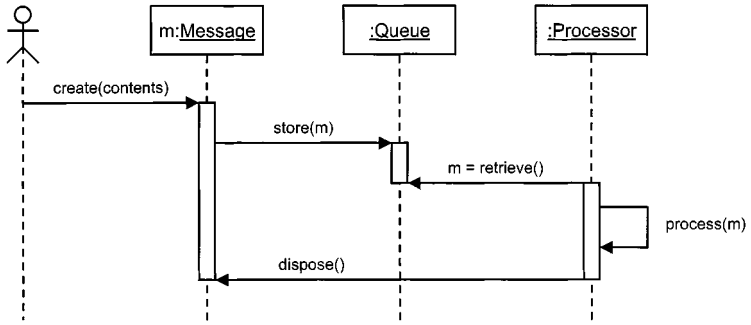


Figure 5. A sequence diagram showing mutual interactions between objects.

- DesignArt_3: the processor class design (from SpecReq_3 and SpecReq_6);
- DesignArt_4: the sequence diagram (from $\text{SpecReq}_1, \ldots, \text{SpecReq}_6$).

A key point is that the queue class design artifact (DesignArt_2) has been assigned to implement the message rate client's requirement (ClientReq_3), which was not implemented during the analysis phase. Suppose that for various reasons the engineers decide to use a fixed length array, then its capacity should be determined based on the peak message arrival rate (300 per second), as well as on the estimated message processing time. Assuming a 0.5 s message-processing time and a redundancy factor of 50%, the array should have a capacity of 225 messages.

Following all the above considerations and the sequence diagram of Figure 5, the following design requirements $\text{DesignReq}_1, \ldots, \text{DesignReq}_7$ are generated:

- DesignReq_1: the message class shall have a create method, with a contents parameter (from DesignArt_1 and DesignArt_4);
- DesignReq_2: the message class shall have a dispose method (from DesignArt_1 and DesignArt_4);
- DesignReq_3: the storage attribute of the queue shall be implemented as a fixed length array with a capacity of 225 messages (from DesignArt_2);
- DesignReq_4: the queue class shall have a store method, with a message parameter (from DesignArt_2 and DesignArt_4);
- DesignReq_5: the queue class shall have a retrieve function, which returns a message (from DesignArt_2 and DesignArt_4);
- DesignReq_6: the processor class shall have a process method, with a message parameter (from DesignArt_3 and DesignArt_4);
- DesignReq_7: the sequence of events shall follow as shown in the class diagram (from DesignArt_4).

Further detailed design of the queue class will yield its internal data management functionality; for brevity, however, we will not elaborate on this here. Once again, a graphical representation of the design artifacts together with their associated requirements may be drawn, as shown in Figure 6.

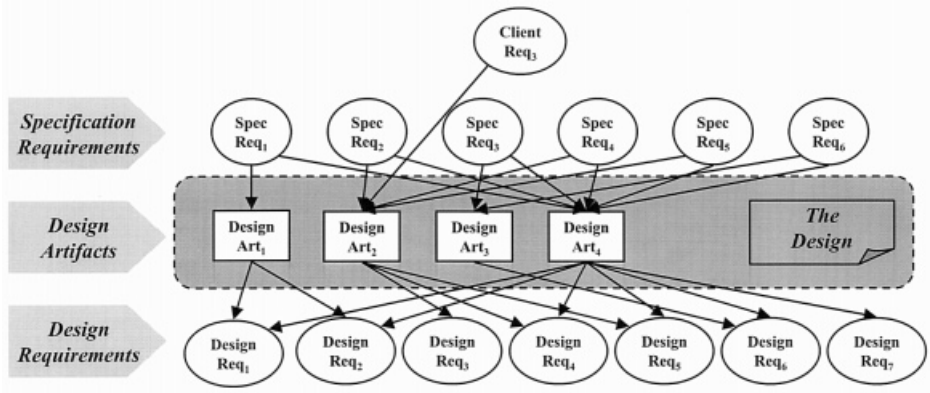


Figure 6. The design artifacts and their associated requirements.

A design document is submitted at the end of the design phase, reflecting the contents of DesignArt_1, ..., DesignArt_4, possibly together with additional documentation.

2.5.4. Implementation

During the implementation phase, code is written to implement the classes of the design using the chosen programming language. We assume, as before, that the code modules, associated with their corresponding classes, are considered to be implementation artifacts (denoted as ImplArt_i), as follows:

- ImplArt_1: the code for the message class (from DesignReq_1, DesignReq_2 and DesignReq_7);
- ImplArt_2: the code for the queue class (from DesignReq_3, DesignReq_4, DesignReq_5 and DesignReq_7);
- ImplArt_3: the code for the processor class (from DesignReq_6 and DesignReq_7).

Finally, we assume that in order to incorporate these classes into an executable program, a 'main program' module is written, constituting another artifact, as follows:

- ImplArt_4: The code for the main program (from DesignReq_7).

3. REQUIREMENTS-DRIVEN CONSTRUCTION PROCESS

3.1. Choice of model

In this section we generalize and formalize the concepts that were informally introduced through the example of the previous section.

In Section 2 we used a simplified waterfall model to describe the course of software construction. We chose the waterfall model because it is more straightforward than, say, the spiral model or incremental

model. However, every software life-cycle model consists of a set of defined steps, each of which is expected to implement the requirements stipulated by previous steps and possibly add additional requirements to be implemented in future steps. Thus, the process described in this section is totally general and can be used with any methodology.

In this section we describe how software is constructed when the complete product is implemented before any maintenance takes place. In Section 4 we extend our results to the more realistic situation of maintenance at any time during the life cycle.

3.2. Relations between artifacts and requirements

As was observed in the example of the previous section, requirements and artifacts play alternating roles in the software construction process, as follows.

- Once a set of requirements is present, a set of artifacts is assigned to implement them. Regardless of the engineering level of abstraction (analysis, design or implementation), the selected artifacts are expected to interpret the requirements and provide an appropriate means to satisfy them.
- In order to satisfy their allocated requirements, artifacts may generate additional requirements to be implemented during subsequent engineering steps. These additional requirements will be allocated to future artifacts, and so on.

We have shown that this relationship between requirements and artifacts can be presented graphically. Paths leading from the client's requirements through alternate layers of artifacts and requirements reveal how requirements propagate through the construction process. We next define a propagation graph as follows.

A *propagation graph* is a directed acyclic graph, consisting of:

- two sets of nodes: *requirements* (represented by ellipses) and *artifacts* (represented by rectangles);
- two sets of directed arcs: *allocation* (leading from a requirement to an artifact) and *generation* (leading from an artifact to a requirement).

Figure 7 depicts the complete propagation graph of Example 1.

The main role played by the propagation graph in the software construction process is to hold the traceability information required when maintenance is to be performed. In practice, however, this information is attached to the artifacts, thereby providing a link between each artifact and its associated requirements. For this purpose we define, for each artifact, two sets of requirements as follows.

Let a be an artifact. Then

R_A(a) is the set of all the requirements r_A, such that there exists an allocation arc leading from r_A to a.

R_G(a) is the set of all the requirements r_G, such that there exists a generation arc leading from a to r_G.

3.3. Generic construction procedure

Based on the definitions of the previous sections, we now define the generic construction procedure incorporating requirements, artifacts and their relationships. The term 'generic' is used because this

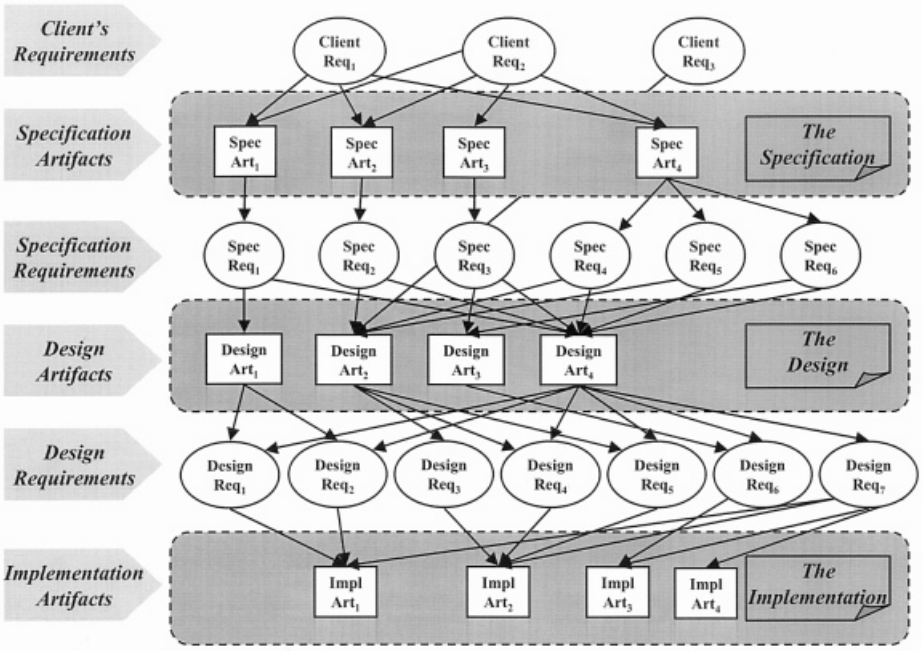


Figure 7. The complete propagation graph of Example 1.

procedure can be applied to any step in any software life-cycle model and development methodology. A step may be interpreted, inter alia, as a phase in the waterfall model, a cycle in a spiral model or a build in an incremental model.

A construction step starts with the selection of a subset of requirements to be satisfied by artifacts engineered during this step. In the waterfall model, as in Example 1, requirements selected at each step are associated with the construction phase. Thus, the client's requirements were selected for the analysis step, then the specification requirements for the design step, and finally the design requirements for the implementation step. However, the personnel may choose a different requirements selection strategy based, for example, on subsystem decomposition, risk analysis or available resources.

The procedure applied at each construction step, formalized in Figure 8, consists of two stages:

- **Stage 1:** the selected requirements are allocated to artifacts. These may either be existing artifacts or newly generated artifacts.
- **Stage 2:** all these artifacts are implemented, thereby satisfying their allocated requirements. During implementation the artifacts may generate additional requirements.

The propagation graph is updated whenever new nodes and arcs are generated.

The personnel use the generic construction procedure of Figure 8 as follows. Initially, the generic construction procedure is applied to the client's requirements R. Thereafter, the personnel select subsets


```
Given a propagation graph P with a set R of requirements and a set A of artifacts:

/* Stage 1: allocation */
    define R' ⊆ R, a subset of requirements to be allocated;
    for each r' in R'
    {
        define a set of artifacts {a1, ..., an} to which r' should be allocated;
                                        /* both existing and newly generated artifacts */
        for each ai, i = 1, ..., n
        {
            if ai is a newly generated artifact then add ai to A and hence to P;
            add an allocation arc in P from r' to ai;
        }
    }

/* Stage 2: construction */
    define A' ⊆ A, the set of all artifacts such that there exist allocation arcs from R' to A';
    for each a' in A'
    {
        implement a', satisfying all the requirements in RA(a');
        for every newly generated requirement r'G
        {
            add r'G to R and hence to P;
            add a generation arc in P from a' to r'G;
        }
    }
```

Figure 8. Generic construction procedure.

of the requirements that have not yet been implemented and apply the generic construction procedure to them until all requirements have been implemented. This process is formalized in the next section.

3.4. Software construction process

When the procedure of Figure 8 is followed at each construction step, it is possible to determine the status of the product at any time.

Prior to any software engineering activity, we have the following:

- a single artifact a_ϕ, comprising the following:
  - an empty set of allocated requirements, $R_A(a_f)$;
  - a set of generated requirements, $R_G(a_\phi)$, the client's requirements;
- an initial propagation graph, with $R_G(a_f)$ as its set of nodes and an empty set of arcs.

After the completion of each construction step, the status of the product is determined by the propagation graph, as follows:

- the current set of all existing requirements R;
- the current set of all existing artifacts A;
- the set of requirements allocated to each artifact $R_A(a)$;
- the set of requirements generated by each artifact $R_G(a)$.

The current status of the product shows which of the requirements have already been implemented (i.e., have a nonempty set of outgoing allocation arcs), as opposed to those that still need to undergo implementation. The personnel now select a subset of the latter requirements and apply the generic construction procedure of Section 3.3 to those requirements. This is repeated until all the requirements have been implemented.

In the next section we will describe how the above process may be modified, and even more efficiently utilized, for the purpose of maintenance.

4. MAINTENANCE

4.1. Maintenance example 1M

The process described in the previous section establishes a methodical way for implementing requirements, and keeping track of their interrelationships and their association with the relevant artifacts. Each construction step starts with a new set of empty artifacts. During its complete life cycle, however, a software product is likely to be changed. This means that requirements are added, changed or (less commonly) deleted. The implementation then needs to be changed accordingly. In order to provide for maintenance, our process of Section 3 needs to be adapted. This will be done after revisiting Example 1 and demonstrating how the maintenance should be handled.

4.1.1. Requirements

Suppose that the software described in Example 1 now needs to process two types of messages: encrypted and unencrypted. We will show how this change is propagated through the artifacts and generated requirements via maintenance. In order to relate the modified example to the original, in what follows we underline each changed artifact or requirement, and newly added artifacts and requirements are assigned the next sequence number. *Italics* are used to highlight changes and additions. The maintenance example will be denoted as Example 1M.

The changes in the client's requirements are expressed by an additional requirement ($\underline{\text{ClientReq}_4}$) and a modified requirement ($\underline{\text{ClientReq}_2}$) as follows (new requirement $\underline{\text{ClientReq}_4}$ should be *read before* modified requirement $\underline{\text{ClientReq}_2}$):

- ClientReq_1: messages arrive randomly;
- $\underline{\text{ClientReq}_2}$: messages, *after decryption*, shall be processed in order of arrival;
- ClientReq_3: the peak rate is 300 messages per second;
- $\underline{\text{ClientReq}_4}$: *messages may be encrypted or unencrypted*.

The immediate impact of these changes is that the propagation graph (of Figure 7) must now contain an additional node, labeled $\underline{\text{ClientReq}_4}$, and a modified node for $\underline{\text{ClientReq}_2}$.

4.1.2. Reanalysis

In contrast to the initial analysis step, the propagation graph at this stage contains a set of existing specification artifacts, namely, $\text{SpecArt} = \{\text{SpecArt}_1, \ldots, \text{SpecArt}_4\}$, together with their allocated

and generated requirements, specified by arcs in the propagation graph. Because ClientReq_1 and ClientReq_3 have not changed, there is no need to reallocate these requirements to artifacts. In order to reveal the impact of $\underline{\text{ClientReq}_2}$ on the analysis artifacts, its original set of outgoing allocation arcs is examined. This shows that $\underline{\text{ClientReq}_2}$ was originally allocated to all four of the specification artifacts, and all of them therefore need to be checked, and revised if needed. As for the new requirement $\underline{\text{ClientReq}_4}$, it needs to be allocated to artifacts (either existing or new ones) at the discretion of the analysis engineer.

Suppose that the reanalysis results in the following decisions:

(i) Every message must contain an encryption flag indicating whether the message was encrypted or not.
(ii) The change in $\underline{\text{ClientReq}_2}$ does not affect any of the class specifications or the class diagram. It is therefore not handled at this stage but is passed on to the next phase.

Thus, at the end of the analysis phase we have the following set of specification requirements (again, only the one underlined requirement has been changed):

- $\underline{\text{SpecReq}_1}$: a Message class shall exist, with contents attribute *and* encryption_flag *attribute*;
- SpecReq_2: a Queue class shall exist, with storage attribute;
- SpecReq_3: a Processor class shall exist;
- SpecReq_4: a queue shall store zero or more messages;
- SpecReq_5: a queue shall retrieve zero or more messages;
- SpecReq_6: a processor shall process zero or more messages.

Thus, the only artifact that has been changed is SpecArt_1, which contains the modified message class. Because neither the allocation nor the generation of the requirements has changed, the propagation graph does not change further as a consequence of the reanalysis.

4.1.3. Redesign

The original set of design artifacts is $\text{DesignArt} = \{\text{DesignArt}_1, \ldots, \text{DesignArt}_4\}$, the three class designs and the sequence diagram. The propagation graph (Figure 7) shows that the original specification requirement SpecReq_1 was allocated to DesignArt_1 (the message class specification) and to DesignArt_4 (the class diagram). Accordingly, only these two artifacts need to be examined with respect to the changed requirement $\underline{\text{SpecReq}_1}$. It turns out that this change to SpecReq_1 (adding a new attribute) is of minor significance for these artifacts, and thus does not generate additional design requirements. As a result, no design artifact or generated requirement has yet been changed.

The client's new requirement $\underline{\text{ClientReq}_4}$ is likely to have a significant impact on the design, because this requirement has not been implemented during reanalysis. Accordingly, its impact on all the design artifacts should be examined. The designer must decide which of the classes should take responsibility for decrypting encrypted messages; according to the changed client's requirement $\underline{\text{ClientReq}_2}$, messages should be decrypted before processing. Suppose that the designer decides that every message is responsible for its own decryption. This decision has the following consequences:

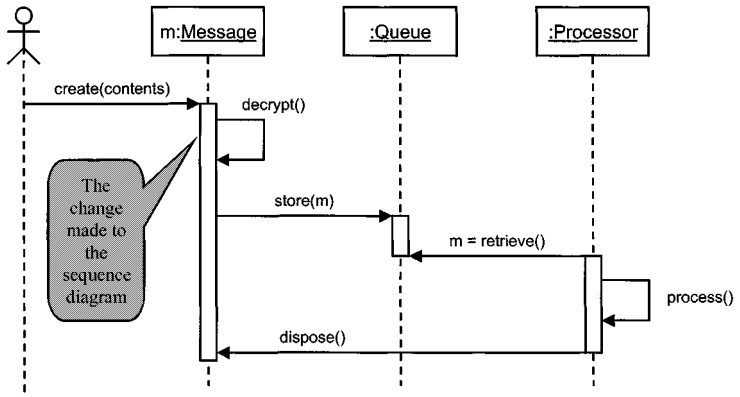


Figure 9. Modified sequence diagram.

(i) The message class should contain a decrypt method (an additional design requirement).
(ii) The sequence of events, as reflected by the sequence diagram, needs to be modified, resulting in a change in existing design artifacts.

The modified sequence diagram, capturing these changes, is shown in Figure 9. The new set of design requirements is now as follows:

- DesignReq_1: the message class shall have a create method, with a contents parameter;
- DesignReq_2: the message class shall have a dispose method;
- DesignReq_3: the storage attribute of the queue shall be implemented as a fixed length array with a capacity of 225 messages;
- DesignReq_4: the queue class shall have a store method with a message parameter;
- DesignReq_5: the queue class shall have a retrieve function which returns a message;
- DesignReq_6: the processor class shall have a process method with a message parameter;
- $\underline{\text{DesignReq}_7}$: the sequence of events shall follow *the modified* sequence diagram;
- $\underline{\text{DesignReq}_8}$: *the message class shall have a* decrypt *method.*

4.1.4. Reimplementation

Final reimplementation is performed by modifying the code for the affected classes. In this case, the impact of both $\underline{\text{DesignReq}_7}$ and $\underline{\text{DesignReq}_8}$ is on only the message class, and therefore only $\underline{\text{ImplArt}_1}$ changes.

4.2. Maintenance process

In the light of the modified Example 1M we now modify the process introduced in Section 3.3 to handle maintenance.

A maintenance step is triggered by one or more changes to the requirements. For example, an existing requirement may have been changed, a new requirement may have been generated, or an existing requirement may have been deleted. With regard to a deletion, our approach is to nullify the contents of the deleted requirement while retaining the requirement itself. Another type of change that may occur is when a requirement is reallocated to a different artifact; this should cause the artifact to which it was formerly allocated to delete the implementation of this requirement. We will show later how this issue is handled in the maintenance process.

For the purpose of keeping track of the changes throughout the maintenance process we will use a marking over the nodes of the propagation graph as follows:

- a marked requirement indicates that the requirement is new or has been changed and needs to be considered for reallocation;
- a marked artifact indicates that this artifact is affected by a change in the requirements and needs to be considered for maintenance.

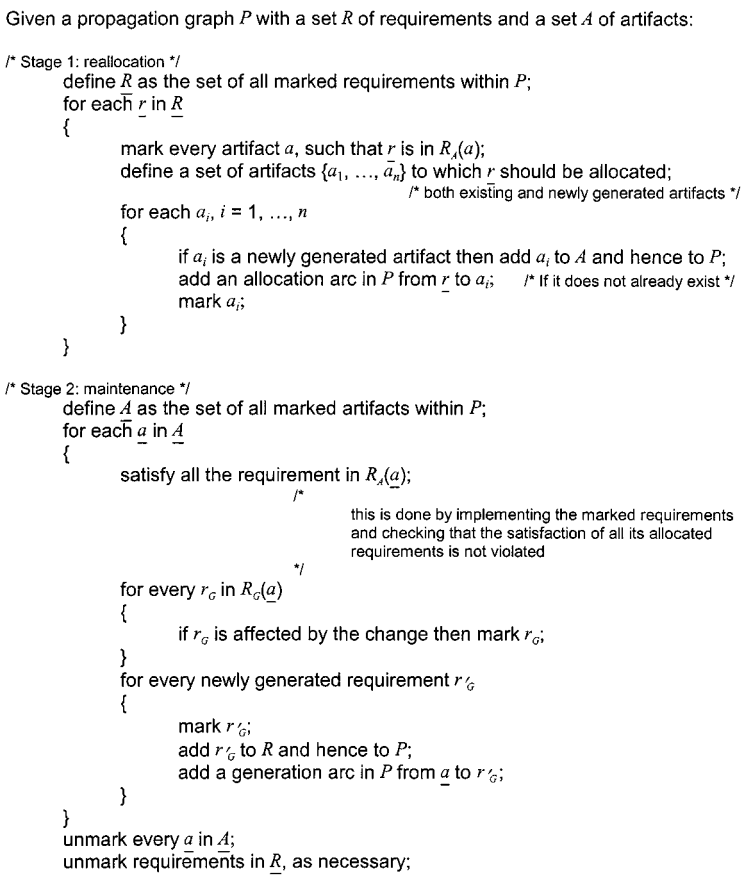
The generic maintenance procedure is formalized in Figure 10. Here we outline key aspects. Maintenance is performed in two alternating stages as follows.

- **Stage 1.** All the marked requirements are examined, resulting in the marking of all their affected artifacts. If any of these requirements appears to need a new artifact, the artifact is generated from scratch, the requirement is allocated to it and then the artifact is marked. At the end of Stage 1 we have a modified propagation graph, with updated allocation arcs between requirements and artifacts.
- **Stage 2.** All the marked artifacts are changed. Every previously generated requirement of these artifacts, which have been affected by the change, is marked. In addition, newly generated requirements are marked. At the end of Stage 2 we have a new version of the propagation graph, with updated generation arcs between artifacts and requirements.

Before the generic procedure can be repeated, the artifacts and requirements of the current step apparently should be unmarked. Although unmarking the currently changed artifacts is obvious, the unmarking of requirements needs more attention. Clearly, the newly generated requirements of the last step should stay marked, because they now carry the change to be propagated to the next steps. However, as for previously marked requirements, we feel that their unmarking should be left to the discretion of the personnel. One reason is that certain requirements should be implemented by more than one set of artifacts; implementing such a requirement by just one set of artifacts does not give sufficient justification for unmarking it. An example for this kind of requirement is an interface requirement between subsystems, which should be implemented by the set of artifacts of both subsystems. Another type of requirement that must not be unmarked is a 'standing' requirement that needs to be considered by every artifact at every step. An example is a naming standard requirement.

The generic maintenance step outlined above is formalized in Figure 10.

A common maintenance decision is to reallocate requirements. For example, suppose that for various reasons the previous allocation of the decryption task to the message class of Example 1M has now changed, and the decryption is now assigned to the process class. In practice, this will cause any implementation of the decrypt method to be removed from the message class. As we explained at the beginning of this section, the relevant requirement ($\underline{\text{DesignReq}}_8$) will be changed to a null requirement and a new requirement, $\underline{\text{DesignReq}}_9$, will now be added, stating that *the process class shall have a*



```
Given a propagation graph P with a set R of requirements and a set A of artifacts:

/* Stage 1: reallocation */
    define R as the set of all marked requirements within P;
    for each r in R
    {
        mark every artifact a, such that r is in R_A(a);
        define a set of artifacts {a_1, ..., a_n} to which r should be allocated;
                                        /* both existing and newly generated artifacts */
        for each a_i, i = 1, ..., n
        {
            if a_i is a newly generated artifact then add a_i to A and hence to P;
            add an allocation arc in P from r to a_i;     /* If it does not already exist */
            mark a_i;
        }
    }

/* Stage 2: maintenance */
    define A as the set of all marked artifacts within P;
    for each a in A
    {
        satisfy all the requirement in R_A(a);
                        /*
                            this is done by implementing the marked requirements
                            and checking that the satisfaction of all its allocated
                            requirements is not violated
                        */
        for every r_G in R_G(a)
        {
            if r_G is affected by the change then mark r_G;
        }
        for every newly generated requirement r'_G
        {
            mark r'_G;
            add r'_G to R and hence to P;
            add a generation arc in P from a to r'_G;
        }
    }
    unmark every a in A;
    unmark requirements in R, as necessary;
```

Figure 10. Generic maintenance procedure.

decrypt method with a message parameter. This new requirement can now be allocated to the process class design artifact. Furthermore, during redesign of the message class, the modified null requirement will be noticed and its implementation will be removed accordingly.

A more complicated case occurs when a requirement that has been allocated to more than one artifact is changed so that it becomes irrelevant to one artifact (and therefore its implementation should be removed from that artifact), but remains valid for the other artifacts. The problem is that the changed requirement is still allocated to the artifact to which it is now irrelevant; this linkage will trigger unnecessary propagation whenever the requirement is further changed in the future. For the sake of simplicity we have not incorporated a formal deallocation mechanism into our procedure. In practice, however, whenever such deallocation is indicated during reconstruction of an artifact, the allocation arc leading from the irrelevant requirement to this artifact should be removed from the propagation graph.

The efficiency of the process depends on the number of marked requirements at the beginning of each iteration of the maintenance process. Therefore, notwithstanding the cases mentioned above, we suggest that in practice all the marked requirements should be unmarked at the end of each iteration, except the following:

- requirements that were marked in the last iteration (this includes affected requirements and newly generated requirements);
- requirements which have not been allocated to any artifact (i.e., have an empty set of outgoing allocation arcs).

4.3. Software construction as maintenance

In Section 1, we mentioned that maintenance may take place during the very early stages of the software life cycle, even before the completion of the first version of the software. In Section 3.4 we put forward a construction process, based upon the propagation graph, and in the previous section we modified the process to handle maintenance. We would now like to show that the simpler (initial) construction case is merely a special case of the maintenance process.

We started the construction process with an initial set of client's requirements, to which we allocated a set of (empty) artifacts. Then these artifacts were implemented, resulting in additional requirements. This was repeated until all the requirements were implemented. In order to do the same in the maintenance context, we need only mark the client's requirements before the first step of the maintenance process. Then, when the generic maintenance procedure is invoked, during its Stage 1 new artifacts are created and requirements are allocated to them. During its Stage 2 these artifacts are implemented, resulting in newly generated requirements. Once the first step has been accomplished, the process continues in the same fashion. However, at this juncture the software is also subject to changes to the requirements, and hence to the artifacts that exist so far. Thus, whether or not the requirements are changed during construction, the process of the previous section is equally applicable.

5. RESULTS, CONCLUSIONS AND FUTURE WORK

We have developed a process for software construction that incorporates maintenance, no matter how early in the life cycle maintenance is needed. The process can be used in conjunction with any development methodology. Our process has two components: a procedure that is applied at each step of the methodology, and a data structure, the propagation graph, that is updated at each step. The propagation graph contains the information needed to determine which artifacts of the product need to be examined if a specific requirement changes, or if a new requirement is introduced.

Three key aspects of our process are:

- generality—the procedure and propagation graph can be utilized irrespective of the software methodology in use;
- intrinsic maintenance—almost no methodologies support changes to the requirements once the product has been delivered, let alone while the product is being developed; and

- uniformity—the same procedure is applied at every step in the software life cycle; no distinction of any kind is drawn between development and maintenance from the viewpoint of changes in the requirements.

In an earlier paper, Tomer and Schach [3] presented the evolution tree life-cycle model, which explicitly describes the way a software product evolves as the requirements change. It is our intention to combine the evolution tree model with the process described in this paper to yield a decision support tool that will enable management to control the way a software product evolves.

REFERENCES

1. Schach SR. *Classical and Object-Oriented Software Engineering with UML and C++*, Fourth edn. WCB McGraw-Hill: New York NY, 1999; 11, 34, 75, 509–510.
2. Yourdon E. *Rise and Resurrection of the American Programmer*. Yourdon Press: Upper Saddle River NJ, 1996.
3. Tomer A, Schach SR. The Evolution Tree: a reengineering-oriented software development model. *Technical Report 99-03*. Computer Science Department, Vanderbilt University, Nashville TN, 1999.
4. Li L, Offutt AJ. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1996; 171–184.
5. IEEE. *A Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)*. Institute of Electrical and Electronic Engineers, Inc.: New York NY, 1990; 44.
6. ISO/IEC. *Software Life Cycle Processes (ISO/IEC Std 12207)*. International Organization for Standardization, International Electrotechnical Commission: Geneva, Switzerland, 1995; 24.
7. Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman: Reading MA, 1999; 41–59, 85–91.
8. RequisitePro. *RequisitePro Product Information*. Rational Software Corporation: Cupertino CA, 1999. Also available at URL: http://www.rational.com/products/reqpro/ [18 August 1999].
9. Tomer A. Implementing specifications using logic with inheritance. *Ph.D Thesis*. Department of Computing, Imperial College of Science Technology and Medicine, University of London, London, UK, 1992; 52–73.
10. Tomer A, Hogger CJ. A logic approach to knowledge-based engineering. In *Applications of Artificial Intelligence in Engineering IX*, Rzevski G, Adey RA, Russell DW (eds.). Computational Mechanics Publications: Southampton, UK, 1994; 237–245.
11. Jacobson I, Booch G, Rumbaugh J. *The Unified Software Development Process*. Addison Wesley Longman: Reading MA, 1999; 443.
12. Lindvall M, Sandahl K. Practical implication of traceability. *Software—Practice and Experience* 1996; **26**(10):1161–1180.
13. Kotonya G, Sommerville I. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons Ltd.: Chichester, UK, 1997; 114, 128–134.
14. Thompson SM. Configuration management—keeping it all together. *BT Technology Journal* 1997; **15**(3):48–60.
15. Paulk MC, Weber CV, Garcia S, Chrissis MB, Bush M. Key practices of the capability maturity model, version 1.1. *Technical Report CMU/SEI-93-TR-025*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 1993; L2-1–L2-10, L2-75–L2-79.

AUTHORS' BIOGRAPHIES

Stephen R. Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University in Nashville in Tennessee. He is the author of over 95 refereed research papers. Steve has written seven software engineering textbooks, including *Classical and Object-Oriented Software Engineering with UML and Java, Fourth Edition*. He consults internationally on software engineering topics. Steve's research interests are in object-oriented software engineering and software maintenance. He obtained his Ph.D. from the University of Cape Town in South Africa. Email: srs@vuse.vanderbilt.edu

Amir Tomer is the Director of Software Engineering Processes at RAFAEL, Israel, with whom he has been for the last 18 years, holding a variety of software engineering positions, both technical and managerial. His B.Sc. and M.Sc. degrees in Computer Science are from the Technion, Israel, and his Ph.D. in Computing from Imperial College, London, UK. Amir also teaches software engineering at the Technion. The work reported in this paper was done while visiting the Technion on sabbatical leave from RAFAEL. Email: tomera@rafael.co.il